A Characterization of NCk by First Order Functional Programs

Jean-Yves Marion, Romain Péchoux

A Characterization of NC^k by First Order Functional Programs

Jean-Yves Marion and Romain Péchoux

Loria-INPL, École Nationale Supérieure des Mines de Nancy, B.P. 239, 54506 Vandœuvre-lès-Nancy Cedex, France, Jean-Yves.Marion@loria.fr
Romain.Pechoux@loria.fr

Abstract. This paper is part of a research on static analysis in order to predict program resources and belongs to the implicit computational complexity line of research. It presents intrinsic characterizations of the classes of functions, which are computable in NC^k, that is by a uniform, poly-logarithmic depth and polynomial size family of circuits, using first order functional programs. Our characterizations are new in terms of first order functional programming language and extend the characterization of NC^1 in [9]. These characterizations are obtained using a complexity measure, the sup-interpretation, which gives upper bounds on the size of computed values and captures a lot of program schemas.

1 Introduction

Our work is related to machine independent characterizations of functional complexity classes initiated by Cobham's work [14] and studied by the *Implicit computational complexity (ICC)* community, including safe recursion of Bellantoni and Cook [4], data tiering of Leivant [23], linear type disciplines by Girard et al. [17, 18], Lafont [22], Baillot-Mogbil [2], Gaboardi-Ronchi Della Rocca [16] and Hofmann [19] and studies on the complexity of imperative programs using matrix algebra by Kristiansen-Jones [21] and Niggl-Wunderlich [28]. Traditional results of the ICC focus on capturing all functions of a complexity class and we should call this approach extensional whereas our approach, which tries to characterize a class of programs, which represents functions in some complexity classes, as large as possible, is rather intensional. In other words, we try to delineate a broad class of programs using a certain amount of resources.

Our approach relies on methods combining term rewriting systems and interpretation methods for proving complexity upper bounds by static analysis. It consists in assigning a function from real numbers to real numbers to some symbols of a program. Such an assignment is called a sup-interpretation if it satisfies some specific semantics properties introduced in [27]. Basically, a sup-interpretation provides upper bounds on the size of computed values. Sup-interpretation is a generalization of the notion of quasi-interpretation of [10]. The problem of finding a quasi-interpretation or sup-interpretation of a given program, called synthesis problem, is crucial for potential applications of the

method. It consists in automatically finding an interpretation of a program in order to determine an upper bound on its complexity. It was demonstrated in [1, 8] that the synthesis problem is decidable in exponential time for small classes of polynomials. Quasi-interpretations and sup-interpretations have already been used to capture the sets of functions computable in polynomial time and space [26, 7] and capture a broad class of algorithms, including greedy algorithms and dynamic programming algorithms. Consequently, it is a challenge to study whether this approach can be adapted to characterize small parallel complexity classes. Parallel algorithms are difficult to design. Employing the sup-interpretation method leads to delineate efficient parallel programs amenable to circuit computing. Designing parallel implementations of first order functional programs with interpretation methods for proving complexity bounds, might be thus viable in the near future.

A circuit C_n is a directed acyclic graph built up from Boolean gates *And*, *Or* and *Not*. Each gate has an in-degree less or equal to two and an out-degree equal to one. A circuit has n input nodes and $g(n)$ output nodes, where $g(n) = O(n^c)$, for some constant $c \geq 1$. Thus, a circuit C_n computes a function $f_n : \{0,1\}^n \to \{0,1\}^{g(n)}$. A family of circuits is a sequence of circuits $C = (C_n)_n$, which computes a family of finite functions $(f_n)_n$ over $\{0,1\}^*$. A function f is computed by a family of circuits $(C_n)_n$ if the restriction of f to inputs of size n is computed by C_n. A uniformity condition ensures that there is a procedure which, given n, produces a description of the circuit C_n. Such a condition is introduced to ensure that a family of circuits computes a reasonable function. All along, we shall consider U_{E^*}-uniform family of circuits defined in [29]. The complexity of a circuit depends on its depth (the longest path from an input to an output gate) and its size (the number of gates). The class NC^k is the class of functions computable by a U_{E^*}-uniform family of circuits of size bounded by $O(n^d)$, for some constant d, and depth bounded by $O(\log^k(n))$. Intuitively, it corresponds to the class of functions computed in poly-logarithmic time with a polynomial number of processors. Following [3], the main motivation in the introduction of such classes was the search for separation results: "NC^1 is the at the frontier where we obtain interesting separation results". NC^1 contains binary addition, substraction, prefix sum of associative operators. Buss [11] has demonstrated that the evaluation of boolean formulas is a complete problem for NC^1. A lot of natural algorithms belong to the distinct levels of the NC^k hierarchy. In particular, the reachability problem in a graph or the search for a minimum covering tree in a graph are two problems in NC^2.

In this paper, we define a restricted class of first order functional programs, called *fraternal and arboreal* programs, using the notion of sup-interpretation of [27]. We demonstrate that functions, which are computable by these programs at some rank k, are exactly the functions computed in NC^k. This result generalizes the characterization of NC^1 established in [9]. To our knowledge, these are the first results, which connect small parallel complexity classes and first order functional programs.

2 First order functional programs

2.1 Syntax of programs

We define a generic first order functional programming language. The vocabulary $\Sigma = \langle Var, Cns, Op, Fct \rangle$ is composed of four disjoint sets of symbols. The arity of a symbol is the number n of arguments that it takes. A program $\mathbf{p}$ consists in a vocabulary and a set of rules $\mathcal{R}$ defined by the following grammar:

$$\begin{array}{lllll}
\texttt{(Values)} & \mathcal{T}(Cns) \ni \mathbf{v} & ::= & \mathbf{c} \mid \mathbf{c}(\mathbf{v}_1, \cdots, \mathbf{v}_n) \\
\texttt{(Patterns)} & \mathcal{T}(Var, Cns) \ni p & ::= & \mathbf{c} \mid x \mid \mathbf{c}(p_1, \cdots, p_n) \\
\texttt{(Expressions)} & \mathcal{T}(Var, Cns, Op, Fct) \ni e & ::= & \mathbf{c} \mid x \mid \mathbf{c}(e_1, \cdots, e_n) \\
 & & & \mid \mathbf{op}(e_1, \cdots, e_n) \mid \mathtt{f}(e_1, \cdots, e_n) \\
\texttt{(Rules)} & \mathcal{R} \ni r & ::= & \mathtt{f}(p_1, \cdots, p_n) \to e
\end{array}$$

where $x \in Var$ is a variable, $\mathbf{c} \in Cns$ is a constructor symbol, $\mathbf{op} \in Op$ is an operator, $\mathtt{f} \in Fct$ is a function symbol, $p_1, \cdots, p_n \in \mathcal{T}(Var, Cns)$ are patterns and $e_1, \cdots, e_n \in \mathcal{T}(Var, Cns, Op, Fct)$ are expressions. The program's main function symbol is the first function symbol in the program's list of rules.

Throughout the paper, we only consider orthogonal programs having disjoint and linear rule patterns. Consequently, each program is confluent [20]. We will use the notation $\overline{e}$ to represent a sequence of expressions, that is $\overline{e} = e_1, \ldots, e_n$.

2.2 Semantics

The domain of computation of a program $\mathbf{p}$ is the constructor algebra $\mathtt{Values} = \mathcal{T}(Cns)$. Set $\mathtt{Values}^* = \mathtt{Values} \cup \{\mathbf{Err}\}$, where $\mathbf{Err}$ is a special symbol associated to runtime errors. An operator $\mathbf{op}$ of arity n is interpreted by a function $[\![\mathbf{op}]\!]$ from $\mathtt{Values}^n$ to $\mathtt{Values}^*$. Operators are essentially basic partial functions like destructors or characteristic functions of predicates like $=$.

Set $\mathtt{Values}^\# = \mathtt{Values} \cup \{\mathbf{Err}, \bot\}$, where $\bot$ means that a program is non-terminating. Given a program $\mathbf{p}$ of vocabulary $\langle Var, Cns, Op, Fct \rangle$ and an expression $e \in \mathcal{T}(Cns, Op, Fct)$, the computation of e, noted $[\![e]\!]$, is defined by $[\![e]\!] = \mathbf{w}$ iff $e \xrightarrow{*} \mathbf{w}$ and $\mathbf{w} \in \mathtt{Values}^*$, otherwise $[\![e]\!] = \bot$, where $\xrightarrow{*}$ is the reflexive and transitive closure of the rewriting relation $\to$ induced by the rules of $\mathcal{R}$. By definition, if no rule is applicable, then an error occurs and $[\![e]\!] = \mathbf{Err}$. A program of main function symbol $\mathtt{f}$ computes a partial function $\phi : \mathtt{Values}^n \to \mathtt{Values}^*$ defined by $\forall \mathbf{u}_1, \cdots, \mathbf{u}_n \in \mathtt{Values},\ \phi(\mathbf{u}_1, \cdots, \mathbf{u}_n) = w$ iff $[\![\mathtt{f}(\mathbf{u}_1, \cdots, \mathbf{u}_n)]\!] = w$.

Definition 1 (Size). *The size of an expression e is defined by $|e| = 0$, if e is a 0-arity symbol, and $|b(e_1, \cdots, e_n)| = \sum_{i \in \{1,\ldots,n\}} |e_i| + 1$, if $e = b(e_1, \cdots, e_n)$.*

Example 1. Consider the following program which computes the logarithm function over binary numbers using the constructor symbols $\{\mathbf{0}, \mathbf{1}, \epsilon\}$:

$$\begin{array}{rl}
\mathtt{f}(x) & \to \mathbf{rev}(\log(x)) \\
\log(\mathbf{i}(x)) & \to \mathbf{if}(\mathbf{Msp}(\mathbf{Fh}(\mathbf{i}(x)), \mathbf{Sh}(\mathbf{i}(x))), \mathbf{0}(\log(\mathbf{Fh}(\mathbf{i}(x)))), \mathbf{1}(\log(\mathbf{Fh}(\mathbf{i}(x))))) \\
\log(\epsilon) & \to \epsilon
\end{array}$$

where $\mathbf{if}(u, v, w)$ is an operator, which outputs v or w depending on whether u is equal to $\mathbf{1}(\epsilon)$ or $\mathbf{0}(\epsilon)$, **rev** is an operator, which reverses a binary value given as input, **Msp** is an operator, which returns $\mathbf{1}(\epsilon)$ if the leftmost $|x| \ominus |y|$ bits of x (where $\forall x, y \in \mathbb{N}, x \ominus y = 0$ if $y > x$ and $x - y$ otherwise) are equal to the empty word ϵ and returns $\mathbf{0}(\epsilon)$ otherwise, **Fh** is an operator, which outputs the leftmost $\lfloor |x|/2 \rfloor$ bits of x, and **Sh** is an operator, which outputs the rightmost $\lceil |x|/2 \rceil$ bits of x.

The algorithm tests whether the number $\lfloor |x|/2 \rfloor$ of leftmost bits is equal to the number $\lceil |x|/2 \rceil$ of rightmost bits in the input x using the operator **Msp**. In this case, the last digit of the logarithm is a 0, otherwise it is a 1. Finally, the computation is performed by applying a recursive call over half of the input digits and the result is obtained by reversing the output, using the operator **rev**. For any binary value $\mathbf{v}$, we have $[\![\log(\mathbf{v})]\!] = \mathbf{u}$, where $\mathbf{u}$ is the value representing the binary logarithm of the input value $\mathbf{v}$. For example, $[\![\log(\mathbf{1}(\mathbf{0}(\mathbf{0}(\mathbf{0}(\epsilon)))))]\!] = \mathbf{1}(\mathbf{0}(\mathbf{0}(\epsilon)))$.

2.3 Call-tree

We now describe the notion of call-tree which is a representation of a program state transition sequences induced by the rewrite relation $\rightarrow$ using a call-by-value strategy. In this paper, the notion of call-tree allows to control the successive function calls corresponding to a recursive rule. First, we define the notions of context and substitution. A *context* is an expression $\mathsf{C}[\diamond_1, \cdots, \diamond_r]$ containing one occurrence of each $\diamond_i$, with $\diamond_i$ new variables which do not appear in Σ. A substitution is a finite mapping from variables to $\mathcal{T}(\mathit{Var}, \mathit{Cns}, \mathit{Op}, \mathit{Fct})$. The substitution of each $\diamond_i$ by an expression d_i in the context $\mathsf{C}[\diamond_1, \cdots, \diamond_r]$ is noted $\mathsf{C}[d_1, \cdots, d_r]$. A ground substitution σ is a mapping from variables to $\mathtt{Values}$. Throughout the paper, we use the symbol σ and the word "substitution" to denote a ground substitution. The application of a substitution σ to an expression (or a sequence of expressions) e is noted $e\sigma$.

Definition 2. *Suppose that we have a program $\boldsymbol{p}$. A* state $\langle \mathtt{f}, \boldsymbol{u}_1, \cdots, \boldsymbol{u}_n \rangle$ *of $\boldsymbol{p}$ is a tuple where* $\mathtt{f}$ *is a function symbol of arity n and $\boldsymbol{u}_1, \cdots, \boldsymbol{u}_n$ are values of* $\mathtt{Values}^*$.

There is state transition*, noted $\eta_1 \rightsquigarrow \eta_2$, between two states $\eta_1 = \langle \mathtt{f}, \boldsymbol{u}_1, \ldots, \boldsymbol{u}_n \rangle$ and $\eta_2 = \langle \mathsf{g}, \boldsymbol{v}_1, \cdots, \boldsymbol{v}_m \rangle$ if there are a rule $\mathtt{f}(p_1, \cdots, p_n) \rightarrow e$ of $\boldsymbol{p}$, a substitution σ, a context $\mathsf{C}[-]$ and expressions $e_1, \cdots, e_m$ such that $\forall i \in \{1, n\}$ $p_i\sigma = \boldsymbol{u}_i$, $\forall j \in \{1, m\}$, $[\![e_j\sigma]\!] = \boldsymbol{v}_j$ and $e = \mathsf{C}[\mathsf{g}(e_1, \cdots, e_m)]$. We write $\overset{*}{\rightsquigarrow}$ to denote the reflexive and transitive closure of $\rightsquigarrow$. A call-tree of $\boldsymbol{p}$ of root $\langle \mathtt{f}, \boldsymbol{u}_1, \cdots, \boldsymbol{u}_n \rangle$ is the following tree:*

- *the root is the node labeled by the state $\langle \mathtt{f}, \boldsymbol{u}_1, \cdots, \boldsymbol{u}_n \rangle$.*
- *the nodes are labeled by states of $\{\eta \mid \langle \mathtt{f}, \boldsymbol{u}_1, \cdots, \boldsymbol{u}_n \rangle \overset{*}{\rightsquigarrow} \eta\}$,*
- *there is an edge between two nodes η_1 and η_2 if there is a transition between both states which label the nodes (i.e. $\eta_1 \rightsquigarrow \eta_2$).*

A branch of the call-tree is a sequence of states of the call-tree $\eta_1, \cdots, \eta_k$ *such that* $\eta_1 \rightsquigarrow \eta_2 \ldots \rightsquigarrow \eta_{k-1} \rightsquigarrow \eta_k$. *Given a branch* B *of a call-tree, the depth of the branch* $\mathtt{depth}(B)$ *is the number of states in the branch, i.e. if* $B = \eta_1, \ldots, \eta_{i-1}, \eta_i$, *then* $\mathtt{depth}(B) = i$.

Notice that a call-tree may be infinite if it corresponds a non-terminating program.

2.4 Fraternity

The fraternity is the main syntactic notion, we use in order to restrict the computational power of considered programs.

Definition 3. *Given a program* $\boldsymbol{p}$, *the* precedence $\geq_{Fct}$ *is defined on function symbols by* $\mathtt{f} \geq_{Fct} \mathtt{g}$ *if there is a rule* $\mathtt{f}(\overline{p}) \rightarrow \mathsf{C}[\mathtt{g}(\overline{e})]$ *in* $\boldsymbol{p}$. *The reflexive and transitive closure of* $\geq_{Fct}$ *is also noted* $\geq_{Fct}$. *Define* $\approx_{Fct}$ *by* $\mathtt{f} \approx_{Fct} \mathtt{g}$ *iff* $\mathtt{f} \geq_{Fct} \mathtt{g}$ *and* $\mathtt{g} \geq_{Fct} \mathtt{f}$ *and define* $>_{Fct}$ *by* $\mathtt{f} >_{Fct} \mathtt{g}$ *iff* $\mathtt{f} \geq_{Fct} \mathtt{g}$ *and not* $\mathtt{g} \geq_{Fct} \mathtt{f}$. *We extend the precedence to operators and constructor symbols by* $\forall \mathtt{f} \in Fct$, $\forall b \in Cns \cup Op$, $\mathtt{f} >_{Fct} b$.

Definition 4. *Given a program* $\boldsymbol{p}$, *an expression* $\mathsf{C}[\mathtt{g}_1(\overline{e_1}), \ldots, \mathtt{g}_r(\overline{e_r})]$ *is a* fraternity *activated by* $\mathtt{f}(p_1, \cdots, p_n)$ *if:*

1. $\mathtt{f}(p_1, \cdots, p_n) \rightarrow \mathsf{C}[\mathtt{g}_1(\overline{e_1}), \ldots, \mathtt{g}_r(\overline{e_r})]$ *is a rule of* $\boldsymbol{p}$,
2. *For each* $i \in \{1, r\}$, $\mathtt{g}_i \approx_{Fct} \mathtt{f}$,
3. *For every symbol* b *in the context* $\mathsf{C}[\diamond_1, \cdots, \diamond_r]$, $\mathtt{f} >_{Fct} b$.

Notice that a fraternity corresponds to a recursive rule.

Example 2. $\mathbf{if}(\mathbf{Msp}(\mathbf{Fh}(\mathbf{i}(x)), \mathbf{Sh}(\mathbf{i}(x))), \mathbf{0}(\log(\mathbf{Fh}(\mathbf{i}(x)))), \mathbf{1}(\log(\mathbf{Fh}(\mathbf{i}(x)))))$ is the only fraternity in the program of example 1. It is activated by $\log(\mathbf{i}(x))$ by taking $\mathsf{C}[\diamond_1, \diamond_2] = \mathbf{if}(\mathbf{Msp}(\mathbf{Fh}(\mathbf{i}(x)), \mathbf{Sh}(\mathbf{i}(x))), \mathbf{0}(\diamond_1), \mathbf{1}(\diamond_2))$ since $\log \approx_{Fct} \log$.

3 Sup-interpretations

3.1 Monotonic, polynomial and additive partial assignments

Definition 5. *A partial assignment* $\mathcal{I}$ *is a partial mapping from the vocabulary* Σ *such that, for each symbol* b *of arity* n *in the domain of* $\mathcal{I}$, *it yields a partial function* $\mathcal{I}(b) : (\mathbb{R}^+)^n \longmapsto \mathbb{R}^+$, *where* $\mathbb{R}^+$ *is the set of non-negative real numbers. The domain of a partial assignment* $\mathcal{I}$ *is noted* $dom(\mathcal{I})$ *and satisfies* $Cns \cup Op \subseteq dom(\mathcal{I})$.

A partial assignment $\mathcal{I}$ *is monotonic if for each symbol* $b \in dom(\mathcal{I})$, *we have* $\forall i \in \{1, \ldots, n\}$, $\forall X_i, Y_i \in \mathbb{R}^+$, $X_i \geq Y_i \Rightarrow \mathcal{I}(b)(X_1, \cdots, X_n) \geq \mathcal{I}(b)(Y_1, \cdots, Y_n)$.

A partial assignment $\mathcal{I}$ *is* polynomial *if for each symbol* b *of* $dom(\mathcal{I})$, $\mathcal{I}(b)$ *is a max-polynomial function ranging over* $\mathbb{R}^+$. *That is,* $\mathcal{I}(b) = \max(P_1, \ldots, P_k)$, *with* P_j *polynomials.*

A partial assignment is additive *if the assignment of each constructor symbol* $\mathbf{c}$ *of arity* n *is of the shape* $\mathcal{I}(\mathbf{c})(X_1, \cdots, X_n) = \sum_{i=1}^{n} X_i + \alpha_{\mathbf{c}}$, *with* $\alpha_{\mathbf{c}} \geq 1$, *whether* $n > 0$, *and* $\mathcal{I}(\mathbf{c}) = 0$ *otherwise.*

If each function symbol of a given expression e having m variables $x_1, \cdots, x_m$ belongs to $\mathrm{dom}(\mathcal{I})$ then, given m fresh variables $X_1, \cdots, X_m$ ranging over $\mathbb{R}^+$, we define the homomorphic extension $\mathcal{I}^*(e)$ of the assignment $\mathcal{I}$ inductively by:

1. If x_i is a variable of *Var*, then $\mathcal{I}^*(x_i) = X_i$
2. If b is a symbol of Σ of arity 0, then $\mathcal{I}^*(b) = \mathcal{I}(b)$.
3. If b is a symbol of arity $n > 0$ and $e_1, \cdots, e_n$ are expressions, then

$$\mathcal{I}^*(b(e_1, \cdots, e_n)) = \mathcal{I}(b)(\mathcal{I}^*(e_1), \ldots, \mathcal{I}^*(e_n))$$

Given a sequence $\overline{e} = e_1, \cdots, e_n$, we will sometimes use the notation $\mathcal{I}^*(\overline{e})$ to denote $\mathcal{I}^*(e_1), \ldots, \mathcal{I}^*(e_n)$.

3.2 Sup-interpretations

Definition 6. *A sup-interpretation is a partial assignment θ which satisfies the three conditions below:*

1. *θ is a monotonic assignment.*
2. *For each $\boldsymbol{v} \in \mathtt{Values}$, $\theta^*(\boldsymbol{v}) \geq |\boldsymbol{v}|$*
3. *For each symbol $b \in \mathrm{dom}(\theta)$ of arity n and for each value $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n$ of $\mathtt{Values}$, if $[\![b(\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n)]\!] \in \mathtt{Values}$, then*

$$\theta^*(b(\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n)) \geq \theta^*([\![b(\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n)]\!])$$

A sup-interpretation is additive (respectively polynomial) if it is an additive (respectively polynomial) assignment.

Notice that if a sup-interpretation is an additive assignment then the second condition of the above definition is always satisfied. Intuitively, the sup-interpretation is a special program interpretation which bounds from above the output size of the function denoted by the program, as demonstrated in the following lemma:

Lemma 1 ([27]). *Given a sup-interpretation θ and an expression e defined over $\mathrm{dom}(\theta)$, if $[\![e]\!] \in \mathtt{Values}$ then we have $|[\![e]\!]| \leq \theta^*([\![e]\!]) \leq \theta^*(e)$*

Example 3. Consider the program of example 1. Define the additive assignment θ by $\theta(\mathbf{1})(X) = \theta(\mathbf{0})(X) = X + 1$, $\theta(\epsilon) = 0$, $\theta(\mathbf{Msp})(X, Y) = X \ominus Y = \max(X - Y, 0)$, $\theta(\mathbf{Fh})(X) = X/2$ and $\theta(\mathbf{Sh})(X) = X/2$. We claim that θ is an additive and polynomial sup-interpretation. Indeed, all these functions are monotonic. Moreover, for every binary value $\mathbf{v}$, we have $\theta^*(\mathbf{v}) = |\mathbf{v}|$ since the sup-interpretation of a value is equal to its size. Finally, such an assignment satisfies the third condition of the above definition. In particular, we check that, for every binary values $\mathbf{u}$ and $\mathbf{v}$, if $\mathbf{Msp}(\mathbf{u}, \mathbf{v}) \in \mathtt{Values}$ then we have:

$$\begin{aligned}
\theta^*(\mathbf{Msp}(\mathbf{u}, \mathbf{v})) &= \theta(\mathbf{Msp})(\theta^*(\mathbf{u}), \theta^*(\mathbf{v})) && \text{By definition of assignments} \\
&= \theta^*(\mathbf{u}) \ominus \theta^*(\mathbf{v}) && \text{By definition of } \theta(\mathbf{Msp}) \\
&= |\mathbf{u}| \ominus |\mathbf{v}| && \text{Since } \forall \mathbf{w} \in \mathtt{Values},\ \theta^*(\mathbf{w}) = |\mathbf{w}| \\
&= |[\![\mathbf{Msp}(\mathbf{u}, \mathbf{v})]\!]| && \text{By definition of } \mathbf{Msp} \\
&= \theta^*([\![\mathbf{Msp}(\mathbf{u}, \mathbf{v})]\!]) && \text{Since } \forall \mathbf{w} \in \mathtt{Values},\ \theta^*(\mathbf{w}) = |\mathbf{w}|
\end{aligned}$$

Notice that θ is a partial assignment since it is not defined on the symbols **if** and log. However, we can extend θ by $\theta(\mathbf{if})(X,Y,Z) = \max(Y,Z)$ and $\theta(\log)(X) = X$ in order to obtain a total, additive and polynomial sup-interpretation.

4 Arboreal and fraternal programs

In this section, we give two restrictions on programs (i) the arboreal condition which ensures that the number of successive recursive calls corresponding to a function symbol (in the depth of a call-tree) is bounded logarithmically by the input size (ii) and the fraternal condition which ensures that the size of each computed value is polynomially bounded by the input size.

4.1 Arboreal programs

An arboreal program is a program whose recursion depth (number of successive recursive calls) is bounded logarithmically by the input size. This logarithmic upper bound is obtained by ensuring that some complexity measure, corresponding to the combination of sup-interpretations and motononic and polynomial partial assignments, is divided by a fixed constant $K > 1$ at each recursive call.

Definition 7. *A program $\boldsymbol{p}$ is arboreal iff there are a polynomial and additive sup-interpretation θ, a monotonic and polynomial partial assignment ω and a constant $K > 1$ such that for every fraternity $\mathsf{C}[\mathtt{g}_1(\overline{e_1}), \ldots, \mathtt{g}_r(\overline{e_r})]$ activated by $\mathtt{f}(\overline{p})$, the following conditions are satisfied:*

- *For any substitution σ, $\omega(\mathtt{f})(\theta^*(\overline{p}\sigma)) \geq 1$*
- *For any substitution σ and $\forall i \in \{1, \ldots, r\}$, $\omega(\mathtt{f})(\theta^*(\overline{p}\sigma)) \geq K \times \omega(\mathtt{g}_i)(\theta^*(\overline{e_i}\sigma))$*
- *There is no function symbol $\mathtt{h} \in \overline{e_i}$ such that $\mathtt{h} \approx_{Fct} \mathtt{f}$.*

Example 4. In the program of example 1, there is one fraternity:

$$\mathbf{if}(\mathbf{Msp}(\mathbf{Fh}(\mathbf{i}(x)), \mathbf{Sh}(\mathbf{i}(x))), \mathbf{0}(\log(\mathbf{Fh}(\mathbf{i}(x)))), \mathbf{1}(\log(\mathbf{Fh}(\mathbf{i}(x)))))$$

activated by $\log(\mathbf{i}(x))$. Taking the additive and polynomial sup-interpretation θ of example 3, the polynomial partial assignment $\omega(\log)(X) = X$ and the constant $K = 2$, we check that the program is arboreal:

$$\omega(\log)(\theta^*(\mathbf{i}(x))) = X + 1 \geq 2 \times (X+1)/2 = K \times \omega(\log)(\theta^*(\mathbf{Fh}(\mathbf{i}(x)))) \geq 1$$

Lemma 2. *Assume that $\boldsymbol{p}$ is an arboreal program. Then $\boldsymbol{p}$ is terminating. That is, for every function symbol $\mathtt{f}$ and for any values $\overline{\boldsymbol{u}}$ in $\mathtt{Values}$, $[\![\mathtt{f}(\overline{\boldsymbol{u}})]\!]$ is in $\mathtt{Values}^*$.*

Moreover, for each branch $B = \langle \mathtt{f}, \boldsymbol{u}_1, \cdots, \boldsymbol{u}_n \rangle, \ldots, \langle \mathtt{g}, \boldsymbol{v}_1, \cdots, \boldsymbol{v}_m \rangle$ of a call-tree corresponding to one execution of $\boldsymbol{p}$ and such that $\mathtt{f} \approx_{Fct} \mathtt{g}$, we have:

$$\mathtt{depth}(B) \leq \alpha \times \log(\omega(\mathtt{f})(\theta^*(\boldsymbol{u}_1), \ldots, \theta^*(\boldsymbol{u}_n))), \text{ for some constant } \alpha$$

Proof. Consider an arboreal program $\mathbf{p}$, a call-tree of $\mathbf{p}$ and one of its branch of the shape $\langle \mathtt{f}, \mathbf{u}_1, \cdots, \mathbf{u}_n\rangle \rightsquigarrow \langle \mathtt{g}, \mathbf{v}_1, \cdots, \mathbf{v}_m\rangle$, with $\mathtt{f} \approx_{Fct} \mathtt{g}$. We know, by definition of a call-tree, that there are a rule of the shape $\mathtt{f}(p_1, \cdots, p_n) \rightarrow \mathsf{C}[\mathtt{g}(e_1, \cdots, e_m)]$ and a substitution σ such that $p_i\sigma = \mathbf{u}_i$ and $[\![e_j\sigma]\!] = \mathbf{v}_j$. We obtain:

$$\begin{aligned}\omega(\mathtt{f})(\theta^*(\mathbf{u}_1), \ldots, \theta^*(\mathbf{u}_n)) &= \omega(\mathtt{f})(\theta^*(p_1\sigma), \ldots, \theta^*(p_n\sigma)) \\ &\geq K \times \omega(\mathtt{g}_i)(\theta^*(e_1\sigma), \ldots, \theta^*(e_m\sigma)) \quad \text{By definition 7} \\ &\geq K \times \omega(\mathtt{g}_i)(\theta^*(\mathbf{v}_1), \ldots, \theta^*(\mathbf{v}_m)) \quad \text{By lemma 1}\end{aligned}$$

Applying the same reasoning, we demonstrate, by induction on the depth of a branch, that for each branch $B = \langle \mathtt{f}, \mathbf{u}_1, \cdots, \mathbf{u}_n\rangle \rightsquigarrow \ldots \rightsquigarrow \langle \mathtt{g}, \mathbf{v}_1, \cdots, \mathbf{v}_m\rangle$, with $\mathtt{f} \approx_{Fct} \mathtt{g}$ and $\mathtt{depth}(B) = i$:

$$\omega(\mathtt{f})(\theta^*(\mathbf{u}_1), \ldots, \theta^*(\mathbf{u}_n)) \geq K^i \times \omega(\mathtt{g})(\theta^*(\mathbf{v}_1), \ldots, \theta^*(\mathbf{v}_m))$$

Consequently, the depth is bounded by $\log_K(\omega(\mathtt{f})(\theta^*(\mathbf{u}_1), \ldots, \theta^*(\mathbf{u}_n)))$, because of the first condition of definition 7, whenever $\langle \mathtt{f}, \mathbf{u}_1, \cdots, \mathbf{u}_n\rangle$ is the first state of the considered branch. It remains to combine this result with the equality $\log(x) = \frac{\log_K(x)}{\log_K(2)}$ in order to obtain the required result. Since every branch corresponding to a recursive call has a bounded depth and we are considering confluent programs, the program is terminating. □

4.2 Fraternal programs

Definition 8. *A program $\boldsymbol{p}$ is fraternal if there is a polynomial and additive sup-interpretation θ such that for each fraternity $\mathsf{C}[\mathtt{g}_1(\overline{e_1}), \ldots, \mathtt{g}_r(\overline{e_r})]$ activated by $\mathtt{f}(p_1, \cdots, p_n)$ and for each symbol b of arity m appearing in C or in $\overline{e_j}$, there are constants $\alpha^b_{i,j}, \beta^b_j \in \mathbb{R}^+$ satisfying:*

$$\theta(b)(X_1, \cdots, X_m) = \max_{j \in J}(\sum_{i=1}^{m} \alpha^b_{i,j} \times X_i + \beta^b_j)$$

where J is a finite set of indices.

In other words, a program is fraternal if every symbol in a context or in an argument of a fraternity admits an affinely bounded sup-interpretation.

Example 5. $\mathsf{C}[\log(\mathbf{Fh}(\mathbf{i}(x))), \log(\mathbf{Fh}(\mathbf{i}(x)))]$ is the only fraternity in the program of example 1, where $\mathsf{C}[\diamond_1, \diamond_2] = \mathbf{if}(\mathbf{Msp}(\mathbf{Fh}(\mathbf{i}(x)), \mathbf{Sh}(\mathbf{i}(x))), \mathbf{0}(\diamond_1), \mathbf{1}(\diamond_2))$. Consequently, we have to check that the symbols **if**, **Msp**, **Fh**, **Sh**, **0** and **1** admit affine sup-interpretations. This is the case by taking the polynomial sup-interpretation of example 3 and, consequently, the program is fraternal.

Lemma 3. *Given a sup-interpretation θ and a monotonic and polynomial partial assignment ω for which the program $\boldsymbol{p}$ is arboreal and fraternal, there is a polynomial P such that for each sequence of values $\boldsymbol{u}_1, \cdots, \boldsymbol{u}_n$ and for each function symbol $\mathtt{f}$ of arity n, we have: $|[\![\mathtt{f}(\boldsymbol{u}_1, \cdots, \boldsymbol{u}_n)]\!]| \leq P(\max_{i=1..n}(|\boldsymbol{u}_i|))$*

Proof (Sketch). Lemma 2 states that the number of successive recursive calls occurring in the depth of the call-tree is logarithmically bounded by the input size. By definition 8, the contexts and arguments of a recursive call (fraternity) of a fraternal program are affinely bounded by the input size. Consequently, a logarithmic composition of affinely bounded functions remains polynomially bounded. □

5 Characterizations of $NC^{\mathbf{k}}$ and NC

Similarly to Buss' encoding [11], we represent constructors and destructors (implicitly used in pattern matching definitions) by U_{E^*}-uniform circuits of constant depth and polynomial size. Given such an encoding *code* and a function $\phi : \mathtt{Values}^n \to \mathtt{Values}^*$ computed by some program $\mathbf{p}$, we define a function $\tilde{\phi} : \{0,1\}^* \to \{0,1\}^*$ by $\forall \overline{\mathbf{u}} \in \mathtt{Values}^n\ \tilde{\phi}(\mathit{code}(\overline{\mathbf{u}})) = \mathit{code}(\phi(\overline{\mathbf{u}}))$. A function ϕ of $\mathtt{Values}$ is computable in NC^k relatively to the encoding *code* if and only if $\tilde{\phi}$ is computable by a U_{E^*}-uniform family of circuits in NC^k.

Now, we define a notion of rank in order to make a distinction between the levels of the NC^k hierarchy:

Definition 9. *Given a program $\mathbf{p}$ composed by a vocabulary Σ, a set of rules $\mathcal{R}$ and an encoding code, the rank of a symbol b, $\mathrm{rk}(b)$, and the rank of a symbol b relatively to an expression e, $\nabla(b, e)$, are partial functions ranging over $\mathbb{N}$ and defined by induction over the precedence $\geq_{Fct}$:*

- *If b is a variable or a constructor symbol then $\mathrm{rk}(b) = 0$.*
- *If b is an operator, which can be computed by a U_{E^*}-uniform family of circuits of polynomial size and depth bounded by $\log^k$ relatively to the encoding code, then $\mathrm{rk}(b) = k$.*
- *If b is a function symbol we define its rank relatively to an expression e by:*
  - *If $\forall b' \in e$, $b >_{Fct} b'$ then $\nabla(b, e) = \max_{b' \in e}(\mathrm{rk}(b'))$*
  - *Otherwise $\exists b'' \in e$ such that $b \approx_{Fct} b''$ and $e = b'(e_1, \cdots, e_n)$:*
    - *If $b >_{Fct} b'$ then $\nabla(b, e) = \max(\mathrm{rk}(b') + 1, \nabla(b, e_1), \ldots, \nabla(b, e_n))$*
    - *Otherwise $b \approx_{Fct} b'$ then $\nabla(b, e) = \max(\nabla(b, e_1), \ldots, \nabla(b, e_n)) + 1$*
- *Finally, we define the rank of a function symbol b by:*
  - $\mathrm{rk}(b) = \max_{b(\overline{p}) \to e \in \mathcal{R}}(\nabla(b, e))$

where $b \in e$ means that the symbol b appears in the expression e. The rank of a program is defined to be the highest rank of a symbol of a program.

Example 6. In the program of example 1, the operators **if**, **Fh**, **Sh**, **rev** and **Msp** are of rank 0 since they all belong to NC^0 (Cf.[5] for **Fh**, **Sh** and **Msp**) using an encoding *code* which returns the binary value in $\mathcal{T}(\{\boldsymbol{0}, \boldsymbol{1}, \epsilon\})$ corresponding to a binary string in $\{0,1\}^*$. Consequently, we obtain that:

$$\begin{aligned}
\mathrm{rk}(\log) &= \nabla(\log, \mathbf{if}(\mathbf{Msp}(\mathbf{Fh}(\mathbf{i}(x)), \mathbf{Sh}(\mathbf{i}(x))), \mathbf{0}(\log(\mathbf{Fh}(\mathbf{i}(x)))), \mathbf{1}(\log(\mathbf{Fh}(\mathbf{i}(x))))))\\
&= \max(\mathrm{rk}(\mathbf{if}) + 1, \nabla(\log, \mathbf{0}(\log(\mathbf{Fh}(\mathbf{i}(x))))), \nabla(\log, \mathbf{1}(\log(\mathbf{Fh}(\mathbf{i}(x))))))\\
&= \max(1, \mathrm{rk}(\mathbf{0}) + 1, \mathrm{rk}(\mathbf{1}) + 1, \nabla(\log, \log(\mathbf{Fh}(\mathbf{i}(x)))))\\
&= \max(1, \nabla(\log, \mathbf{Fh}(\mathbf{i}(x))) + 1) = 1
\end{aligned}$$

Theorem 1. *A function ϕ from $\texttt{Values}^n$ to $\texttt{Values}^*$ is computed by a fraternal and arboreal program of rank $k \geq 1$ (resp. $k \in \mathbb{N}$) if and only if ϕ is computable in NC^k (resp. NC).*

Proof (Sketch). We can show, by induction on the rank and the precedence $\geq_{Fct}$, that an arboreal and fraternal program of rank k can be simulated by a U_{E^*}-uniform family of circuits of polynomial size and $\log^k$ depth, using a discriminating circuit of constant depth which, given some inputs, picks the right rule to apply. The $\log^k$ depth relies on a logarithmic number of $\log^{k-1}$ depth circuits compositions by lemma 2. The polynomial size is a consequence of lemma 3. Conversely, we use the characterization of Clote [12] to show the completeness. Clote's algebra is based on two recursion schemas called Concatenation Recursion on Notation (CRN) and Weak bounded Recursion on Notation (WBRN) defined over a function algebra from natural numbers to natural numbers when considering the following initial functions $\texttt{zero}(x) = 0$, $s_0(x) = 2 \times x$, $s_1(x) = 2 \times x + 1$, $\pi^n_k(x_1, \cdots, x_n) = x_k$, $|x| = \lceil \log_2(x+1) \rceil$, $x\#y = 2^{|y| \times |x|}$, $\texttt{bit}(x, i) = \lfloor x/2^i \rfloor \ \texttt{mod}\ 2$ and a function `tree`, which computes alternations of bitwise conjunctions and bitwise disjunctions. All Clote's initial functions can be simulated by operators of rank 1 since they are in AC^0 ([12]). We show that a function of rank k in Clote's algebra can be simulated by an arboreal and fraternal program of rank $k+1$, using divide-and-conquer algorithms (This requirement is needed because of the arboreal property). A difficulty to stress here is that Clote's rank differs from the notion of rank we use because of the function `tree` and the CRN schema. □

6 Comparison with previous works

This work extends the characterization of NC^1 in [9] to the NC^k hierarchy. For that purpose, we have substituted the more general notion of fraternal program to the notion of explicitly fraternal of [9] and we have introduced a notion of rank. In the literature, there are many characterizations of NC^1 using several computational models, like ATM or functions algebra. Compton and Laflamme [15] gave a characterization of NC^1 based on finite functions. Bloch [5] used ramified recurrence schema relying on a divide and conquer strategy to characterize NC^1. Leivant and Marion [25] have established another characterization using ramified recurrence over a specific data structure, well balanced binary trees. We can show that our characterization strictly generalizes the ones of Bloch and Leivant-Marion since they both rely on divide-and-conquer strategies. The characterizations of the NC^k classes by Clote [13], using a bounded recurrence schema à la Cobham, are distinct from our characterizations since they do not rely on a divide and conquer strategy. However, like Cobhams' work, Clote's WBRN schema requires an upper bound on the computed function which is removed from our characterizations. Other characterizations of NC are also provided in [24, 6]. All these purely syntactic characterizations capture a few algorithmic patterns. On the contrary, our work tries to delineate a broad class of algorithms using the semantics notion of sup-interpretation.

References

1. R. Amadio. Synthesis of max-plus quasi-interpretations. *Fundamenta Informaticae, 65(1–2)*, 2005.
2. P. Baillot and V. Mogbil. Soft lambda-calculus: a language for polynomial time computation. In *FOSSACS 2004*, volume 2987 of *LNCS*, pages 27–41. Springer, 2004.
3. D. Barrington, N. Immerman, and H. Straubing. On uniformity within NC. *J. of Computer System Science*, 41(3):274–306, 1990.
4. S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2:97–110, 1992.
5. S. Bloch. Function-algebraic characterizations of log and polylog parallel time. *Computational complexity*, 4(2):175–205, 1994.
6. G. Bonfante, R. Kahle, J.-Y. Marion, and I. Oitavem. Towards an implicit characterization of NC^k. In *CSL'06*, LNCS, 2006.
7. G. Bonfante, J.-Y. Marion, and J.-Y. Moyen. On lexicographic termination ordering with space bound certifications. In *PSI 2001*, volume 2244 of *LNCS*. Springer, Jul 2001.
8. G. Bonfante, J.-Y. Marion, J.-Y. Moyen, and R. Péchoux. Synthesis of quasi-interpretations. *LCC2005, LICS affiliated Workshop*, 2005. `http://hal.inria.fr`.
9. G. Bonfante, J.-Y. Marion, and R. Péchoux. A characterization of alternating log time by first order functional programs. In *LPAR*, volume 4246 of *LNAI*, pages 90–104, 2006.
10. G. Bonfante, J.Y. Marion, and J.Y. Moyen. Quasi-interpretations, a way to control resources. *TCS*, 2007.
11. S. Buss. The boolean formula value problem is in `ALOGTIME`. In *STOC*, pages 123–131, 1987.
12. P. Clote. Sequential, machine-independent characterizations of the parallel complexity classes `ALOGTIME`, AC^k, NC^k and NC. In R. Buss and P. Scott, editors, *Workshop on Feasible Math.*, pages 49–69. Birkhäuser, 1989.
13. P. Clote. Computational models and function algebras. In D. Leivant, editor, *LCC'94*, volume 960 of *LNCS*, pages 98–130, 1995.
14. A. Cobham. The intrinsic computational difficulty of functions. In *Conf. on Logic, Methodology, and Philosophy of Science*, pages 24–30. North-Holland, 1962.
15. K.J. Compton and C. Laflamme. An algebra and a logic for NC. *Inf. Comput.*, 87(1/2):240–262, 1990.
16. M. Gaboardi and S. Ronchi Della Rocca. A soft type assignment system for λ-calculus. *CSL 2007*, 4646:253–267, 2007.
17. J.-Y. Girard. Light linear logic. *Inf. and Comp.*, 143(2):175–204, 1998.
18. J.Y. Girard, A. Scedrov, and P. Scott. Bounded linear logic. *Theoretical Computer Science*, 97(1):1–66, 1992.
19. M. Hofmann. Programming languages capturing complexity classes. *SIGACT News Logic Column 9*, 2000.
20. G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, 1980.
21. L. Kristiansen and N.D. Jones. The flow of data and the complexity of algorithms. In *New Computational Paradigms*, number 3526 in LNCS, pages 263–274, 2005.
22. Y. Lafont. Soft linear logic and polynomial time. *Theoretical Computer Science*, 318(1-2):163–180, 2004.

23. D. Leivant. Predicative recurrence and computational complexity I: Word recurrence and poly-time. In *Feasible Mathematics II*, pages 320–343. Birkhäuser, 1994.
24. D. Leivant. A characterization of *NC* by tree recurrence. In *FOCS'98*, pages 716–724, 1998.
25. D. Leivant and J.-Y. Marion. A characterization of alternating log time by ramified recurrence. *TCS*, 236(1-2):192–208, Apr 2000.
26. J.-Y. Marion and J.-Y. Moyen. Efficient first order functional program interpreter with time bound certifications. In *LPAR*, volume 1955 of *LNCS*, pages 25–42, 2000.
27. J.-Y. Marion and R. Péchoux. Resource analysis by sup-interpretation. In *FLOPS 2006*, volume 3945 of *LNCS*, pages 163–176, 2006.
28. K.-H. Niggl and H. Wunderlich. Certifying polynomial time and linear/polynomial space for imperative programs. *SIAM J. on Computing*. to appear.
29. W. Ruzzo. On uniform circuit complexity. *J. of Computer System Science*, 22(3):365–383, 1981.